VCube-PS: A causal broadcast topic-based publish/subscribe system

João Paulo de Araujo, Luciana Arantes, Elias Duarte Júnior, Luiz Rodrigues, Pierre Sens

VCube-PS: A Causal Broadcast Topic-based Publish/Subscribe System

João Paulo de Araujo[a,*], Luciana Arantes[a], Elias P. Duarte Jr.[b], Luiz A. Rodrigues[c], Pierre Sens[a]

[a] *Sorbonne Université, CNRS, Inria, LIP6 – 4, place Jussieu, 75252, Paris Cedex 5, France*
[b] *Federal University of Paraná (UFPR) – UFPR/Depto. Informática, Caixa Postal 19018, Curitiba, PR, 81531-980, Brazil*
[c] *Western Paraná State University (Unioeste) – R. Universitária, 2069, Universitário, Cascavel, PR, 85819-110, Brazil*

Abstract

In this work we present *VCube-PS*, a topic-based Publish/Subscribe system built on the top of a virtual hypercube-like topology. Membership information and published messages are broadcast to subscribers (members) of a topic group over dynamically built spanning trees rooted at the publisher. For a given topic, the delivery of published messages respects the causal order. *VCube-PS* was implemented on the PeerSim simulator, and experiments are reported including a comparison with the traditional Publish/Subscribe approach that employs a single rooted static spanning-tree for message distribution. Results confirm the efficiency of *VCube-PS* in terms of scalability, latency, number and size of messages.

Keywords:
Publish/Subscribe, Topic-based Pub/Sub, Causal Broadcast, Distributed Spanning Trees, Hypercube-like Topologies

*Corresponding author
Email addresses: joao.araujo@lip6.fr (João Paulo de Araujo), luciana.arantes@lip6.fr (Luciana Arantes), elias@inf.ufpr.br (Elias P. Duarte Jr.), luiz.rodrigues@unioeste.br (Luiz A. Rodrigues), pierre.sens@lip6.fr (Pierre Sens)

1. Introduction

Publish/Subscribe (Pub/Sub) systems consist of a set of *publishers* which are distributed nodes that publish messages that are consumed by *subscribers*. The communication between publishers and subscribers is conducted on an overlay infrastructure, which is generally composed by a set of nodes that organize themselves for ensuring the delivery of published messages to all (and preferably only) subscribers interested in those messages. Hence, publishers and subscribers exchange information asynchronously, without interacting directly [1, 2]. They might even not know each other.

In topic-based Pub/Sub systems, a subscriber can register its interests in one or more topics, and then receives all published messages related to these topics (e.g., Scribe [3], Bayeux [4], DYNATOPS [5], Dynamoth [6], Magnet [7], DRScribe [8], BeaConvey [9], etc.). The advantages of topic-based Pub/Sub systems when compared to content-based systems (see Section 5) are mainly that messages can be statically grouped into topics, the diffusion of messages to subscribers is usually based on multicast groups, and the interface offered to the user is simple. The topic approach is widely used by popular applications including Twitter and Firebase/Google Cloud Messaging, IBM MQ, distributed multiplayer online games, chat systems, and mobile device notification frameworks.

Many topic-based Pub/Sub systems found in the literature are based on per topic broadcast trees built over P2P DHTs [3, 4, 6, 8]. A single multicast tree is associated to each topic composed by both subscribers (resp., brokers) and forwarders, i.e., non-subscribers (resp., non-brokers) of the topic. Therefore, all publish messages related to a topic are broadcast through the same tree. In this work, we call these systems *SRPT* (*Single Root Per Topic*). As they are built over P2P DHTs, they are scalable in terms of the number of subscribers. On the other hand, the maintenance of the one single tree per topic can be costly, particularly when the membership of the system changes [3, 5]. *SRPT* employs multiple forwarders, which are nodes that do not deliver the messages themselves but are employed in the dissemination. Forwarders induce a higher latency and, in the case of a high number of simultaneous publications of a single topic, the root of the tree presents contention problems, becoming a performance bottleneck.

In [10], the authors show that in applications like Twitter, most of the publications are concentrated in few topics: roughly 83% of the analyzed topics have up to 5 published messages and only 0.15% of the topics ("hot

topics") are related to more than 1,000 publishing messages. An example of such applications is multiplayer online combat games in which locations are mapped to topics [11, 12]. During the game, players move towards the same location increasing the publishing load for the topic corresponding to the location, i.e., the location becomes a "hot topic". We argue that *SRPT*-based Pub/Sub systems are not suitable to handle a high publishing load as they present root contention constraints.

We also claim that a topic-based Pub/Sub system must enforce, for a given topic, the causal order of published messages. For instance, in a discussion group, a question published on a group should never be delivered to any subscriber after an answer to that question which was also published in the same group, as the answer is causally related to the question. In other words, if a node publishes a message after it delivers another message, then no node delivers the second message after the first. It is worth emphasizing that causal message ordering is of prime interest to the design of many distributed applications. Examples of them are event notification systems [13], multimedia applications [14, 15], multi-part online games [16], systems that provide distributed replicated causal data consistency [17], distributed snapshots [18], distributed database [19], and shared objects [20]. Specifically for the case of Pub/Sub systems, few works deal with causal ordering [21, 22, 23].

Considering the above discussed points, we propose in this work *VCube-PS*, a non DHT Pub/Sub system, that ensures low latency, and load balancing for publishing messages. It also respects the causal delivery order of published messages of a given topic, which is a crucial feature for several types of Pub/Sub applications. In our system, a published message is sent to all subscribers of a topic by a broadcast protocol that creates a spanning tree composed just by the subscribers, whose root is the publisher. Hence, the root "hot topic" contention problem of *SRPT* does not exist in *VCube-PS* since there is no single root tree per topic, as each node that publishes a message becomes the root of the corresponding spanning tree. Broadcast trees are dynamically built on top of a virtual hypercube-like topology, called *VCube* [24], that presents several logarithmic properties, thus ensuring scalability. Contrarily to *SRPT* Pub/Sub systems and thanks to *VCube*'s properties, both the construction and maintenance of spanning trees by *VCube-PS* have no overhead, even in the presence of subscriber membership changes. In other words, *VCube-PS* locally defines to which nodes the messages need to be forwarded, without the need of routine tables. In the absence of churn, *VCube-PS* does not present forwarder nodes and in the presence of churn,

forwarders are temporary.

It is important to highlight that, contrarily to *SRPT* systems, our target applications are mainly those that present "hot topics" (e.g. multiplayer combat games, company chat groups, etc.).

We implemented *VCube-PS* and two *SRPT*-like Pub/Sub systems on top of the PeerSim simulator [25]. One *SRPT* Pub/Sub is subscriber-based (e.g., Scribe, Magnet, DRScribe) while the second one is broker-based (e.g. Dynatops). In Dynatops, subscribers are connected to brokers based on locality. Results confirm the advantages of using per-publisher dynamically built spanning trees in terms load balancing, latency, number and size of messages metrics.

The rest of the paper is organized as follows. Section 2 gives an overview of *VCube*. Section 3 presents *VCube-PS*'s algorithms to manage topics, order messages, as well as the specification of *VCube-PS*'s algorithms. Section 4 presents evaluation of results conducted on PeerSim simulator. Section 5 discusses related work and, finally, Section 6 concludes the paper.

2. VCube

In *VCube* [24], a node i groups the other $N-1$ nodes in $d = \log_2 N$ clusters forming a *d-VCube*, each cluster s ($s = 1, .., d$) having 2^{s-1} nodes. The ordered list of nodes in each cluster s is defined by function $c_{i,s}$ below, where $\oplus$ is the bitwise exclusive *or* operator (xor).

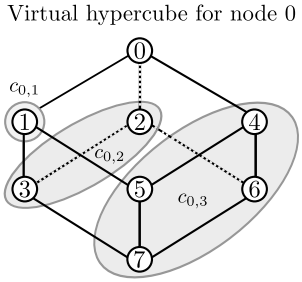
$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i\oplus 2^{s-1},k} \mid k = 1, \ldots, s-1$$

This recursive function can be described as follows. Initially, the first neighbor of node i in cluster s is computed. The identifiers of these two nodes differ only in one bit, the bit that is set to one in 2^{s-1}. Then, the remaining nodes in the cluster are nodes in clusters $1, \ldots, s-1$ of the hypercube neighbor, i.e., $c_{i\oplus 2^{s-1},1}, c_{i\oplus 2^{s-1},2}, \ldots, c_{i\oplus 2^{s-1},s-1}$.

The table of Figure 1 contains, for $N = 8$, the composition of all $c_{i,s}$ of the 3-*VCube*. The same figure also shows node 0's hierarchical cluster-based logical organization in the 3-*VCube*.

3. VCube-PS: Publish/Subscribe System

In this section, we present the topic-based *VCube-PS* Pub/Sub system. We first describe the system model. Then, we describe causal order broad-



The $c_{i,s}$ table for 8 nodes

s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figure 1: *VCube* hierarchical organization.

cast, the use of *causal barriers*, and the per-source FIFO reception order of *VCube-PS*. Finally, we give the algorithms that compose *VCube-PS*.

3.1. System Model and Definitions

We consider a distributed system composed of a finite set of $\Pi = \{0, .., N-1\}$ nodes with $N = 2^d$ nodes, $d > 0$. Each node has a unique identifier (id) and nodes communicate only by message passing. A user of the Pub/Sub system corresponds to a node. Nodes are organized on a logical hypercube.

Nodes communicate by sending and receiving messages. The network is fully connected: each pair of nodes is connected by a bidirectional point-to-point channel and there is no network partitioning. Nodes do not fail and links are reliable. Thus, messages exchanged between any two processes are never lost, corrupted nor duplicated. The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded.

The *source* of a message is the node that broadcasts the message. We distinguish between the arrival of a message (*reception*) at a process and the event that corresponds to the message being delivered to the application/user (*delivery*). Only the latter respects the causal order of published messages.

3.2. Causal and Per-source FIFO Reception Ordering

For each topic, *VCube-PS* enforces the causal order of published messages, implementing, thus, causal broadcast. It also implicitly ensures that for a single publisher, nodes will receive messages in the order they are published.

3.2.1. Causal Ordering

For a given topic t, if a process publishes a message m' after it has delivered a message m, then no process in the system will deliver m after m'. Note

that if a process i never delivers m' (i.e., i leaves the topic before delivering m') or delivers m' but never delivers m (i.e., i was not subscribed to t when m was published), the causal order of published messages is not violated.

In order to implement the causal order of published messages, we apply *causal barriers* [26]. The key advantage of the *causal barrier* approach is that it does not enforce the causal order based on the identifiers of the nodes (per node vector) but by using direct message dependencies, which renders the algorithm more suitable for dealing with node dynamics (subscriptions and unsubscriptions), in comparison to other vector clock-based implements of causal broadcast such as [27] or [28].

Let m and m' be two application messages published for topic t. Message m immediately precedes m' ($m \prec_{im} m'$) if (1) the publishing of m causally precedes the publishing of m' and (2) there exists no message m'' such that the publishing of m causally precedes the publishing of m'', and the publishing of m'' causally precedes the publishing of m'. The *causal barrier* of m (cb_m) consists of the set of messages that are immediate predecessors of m.

Figure 2 shows a distributed system with three nodes (p_0, p_1, and p_2) that have subscribed to the same topic t. Message $m_{s,t,c}$ is the message published by s with sequence number c for topic t. On the left, a timing diagram shows messages being published and delivered; the graph with message dependencies is shown on the right side. We can observe that the delivery of $m_{1,t,1}$ is conditioned by the delivery of $m_{0,t,1}$ ($m_{0,t,1} \prec_{im} m_{1,t,1}$) since p_1 delivered $m_{0,t,1}$ before publishing $m_{1,t,1}$, (i.e., $cb_{m_{1,t,1}} = \{m_{0,t,1}\}$). On the other hand, $m_{1,t,2}$ directly depends on $m_{2,t,1}$ and $m_{1,t,1}$ (i.e., $cb_{m_{1,t,2}} = \{m_{2,t,1}, m_{1,t,1}\}$). Note that since $m_{0,t,1}$ precedes $m_{1,t,1}$ that precedes $m_{1,t,2}$, $m_{0,t,1}$ is an indirect dependency of $m_{1,t,2}$, and was not included, therefore, in $cb_{m_{1,t,2}}$.

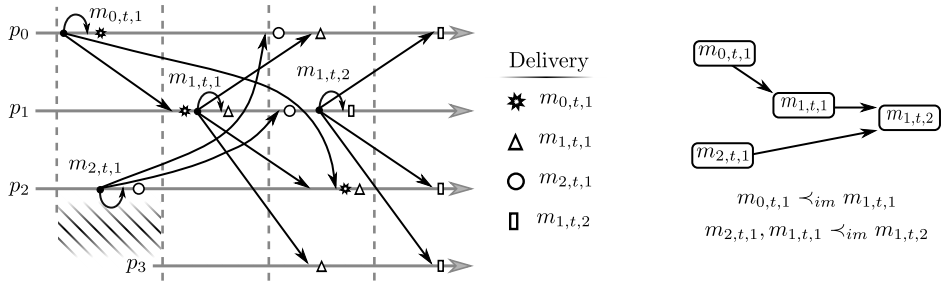


Figure 2: Example of causal barrier.

Now suppose that in the same system shown in Figure 2, p_3 subscribes to

t after messages $m_{2,t,1}$ was published to the other nodes, i.e., node p_3 did not take part in the spanning trees that broadcast $m_{2,t,1}$ and, consequently, in this case, node p_3 will neither receive nor deliver $m_{2,t,1}$. Hence, after having delivered $m_{1,t,1}$, p_3 can deliver $m_{1,t,2}$. Since nodes can dynamically subscribe to or unsubscribe from a topic in *VCube-PS*, our implementation of causal order must distinguish between the case in which a message will be delivered (e.g., $m_{1,t,1}$) from the one that it will never be delivered (e.g., $m_{2,t,1}$ by p_3). To this end, *VCube-PS* guarantees the following property on the FIFO order of messages published on a given topic.

3.2.2. Per-source FIFO Reception Ordering

Messages published by a same publisher are received by subscribers in the same order as they were produced. This order allows a subscriber of t to know that it will never receive some messages previously published, i.e., if $m'_{s,t,c'}$ is the first message that node i receives from s on topic t after it joined t's group, i will never receive $m_{s,t,c}$, $\forall c < c'$.

In *VCube-PS*, per-source FIFO reception order is ensured by the acknowledgment of published messages: a source node broadcasts a new message only after having received all the acknowledgments for the previous message it broadcast. Note that the per-source FIFO reception order is defined in regard to the reception of messages and not delivery, as in the traditional FIFO order definition.

3.3. Algorithms

This section presents *VCube-PS*'s algorithms. Due to the lack of space, proofs are available in an accompanying Technical Report [29]. *VCube-PS* is based on *VCube*, which organizes nodes in a logical hypercube-like topology. Note that in *VCube-PS* nodes do not fail, *VCube-PS* thus exploits *VCube*'s organization but not its failure detection functionality.

Types of Messages, Local Variables and Auxiliary functions: Each message m is uniquely identified by the source (s) and a sequence counter (c). It also carries information about the topic t. Messages can be of type `SUB` (subscription), `UNS` (unsubscribe), `PUB` (publication), and `ACK` (acknowledge). The value of the *data* field depends on the *type* of the message: for `SUB` and `UNS` messages, it holds no information while for `PUB` or `ACK` messages, it respectively holds the published message itself plus causal dependencies (*causal barrier*). MAX_TOPICS is a constant value that limits how many

topics the system can support. The following local variables are kept by every node i:

- *counter*: is a local counter of node i which is incremented at every subscription, unsubscription, or publishing of a message by node i;
- $br\_queue[MAX\_TOPICS]$: each $br\_queue[t]$ is a set of pending messages (`PUB`, `SUB`, or `UNS`) related to the topic t waiting to be broadcast;
- $view[MAX\_TOPICS]$: set of the latest subscription and unsubscription operations of which node i is aware. Each entry $view[t]$ has format $\langle n, o, rc\rangle$ where n is the identity of the node that has joined or left the topic t; o is equal to `SUB` or `UNS` and rc stores the value of the counter of n at the moment the subscription or unsubscription took place;
- $causal\_barrier[MAX\_TOPICS]$: each $causal\_barrier[t]$ keeps information on all messages that are predecessors of the next message that will be published by node i for topic t; the causal barrier consists thus of a set of message identifiers of format $\langle s, c\rangle$ (source and sequence counter).
- *acks*: set of pending `ACK` messages for which i waits confirmation. For each message propagation to its nb children in the spanning tree of a message m identified by $\langle s, t, c\rangle$ received from j, i adds the element $\langle j, nb, \langle s, t, c, mem\rangle\rangle$ to the *acks* set. The set *mem* gathers membership information sent by `ACK` messages;
- *msgs*: set of messages that are being temporarily kept by node i because they have not been delivered yet. Upon delivering m, identified by $\langle s, t, c\rangle$, the latter can be removed from *msgs*;
- $not\_delvs[MAX\_TOPICS]$: each $not\_delvs[t]$ contains a set of messages received by node i for topic t and not yet delivered because their respective *causal barrier* has not been satisfied. Each element has format $\langle s, c, cb\rangle$ where s is the identity of the source node that broadcast the message whose counter is c, and cb corresponds to the *causal barrier* of the message.
- $last\_delvs[MAX\_TOPICS]$: each $last\_delvs[t]$ keeps the identifiers of the last message from each publisher node delivered by node i for topic t. Each element of the set is the tuple $\langle s, c\rangle$ where s is the source identity of the message whose counter is c;
- $first\_rec[MAX\_TOPICS]$: each $first\_rec[t]$ keeps the identifiers of the first message received from each publisher for a topic t. Each element of the set is a tuple $\langle s, c\rangle$.

In the algorithms, the symbol $\perp$ represents a *null* element while the underscore (_) is used to indicate *any* element.

We have defined two auxiliary functions that exploit *VCube* organization and are used to dynamically build broadcast spanning trees:

- CLUSTER(i, j): function that returns the index s of the cluster of node i that contains node j, $(1 \leq s \leq \log_2 N)$. For instance, in Figure 1, CLUSTER$(0, 1) = 1$, CLUSTER$(0, 2) =$ CLUSTER$(0, 3) = 2$, and CLUSTER$(0, 4) =$ CLUSTER$(0, 5) =$ CLUSTER$(0, 6) =$ CLUSTER$(0, 7) = 3$.
- CHILDREN(i, t, h): returns a set with all nodes virtually connected to node i. A child of i is the first node of cluster $c_{i,s}$ which is also a subscriber of topic t; or the first node in $c_{i,s}$ in case of no topic ($t =$ '$*$'). The parameter h can range from 1 to $\log_2 N$. If $h = \log_2 N$, the result set contains the i's children where each child is in $c_{i,s}, s = 1, .., \log_2 N$. For any other value of $h < \log_2 N$, the function returns only a subset of i's children, i.e., those children whose respective cluster number s is smaller or equal to h $(s \leq h)$ For instance, in Figure 1, if $t =$ '$*$', CHILDREN$(0, *, 3) = \{1, 2, 4\}$, CHILDREN$(0, *, 2) = \{1, 2\}$, and CHILDREN$(4, *, 2) = \{5, 6\}$. On the other hand, if only nodes 0, 3, and 4 have joined topic t_1, CHILDREN$(0, t_1, 3) = \{3, 4\}$ and CHILDREN$(4, t_1, 2) = \emptyset$.

Application (User Interface) functions: *VCube-PS* offers an interface consisting of functions SUBSCRIBE(t), UNSUBSCRIBE(t), and PUBLISH(t, m), all presented in Algorithm 1. A node can publish a message related to a topic if it is currently a subscriber of this topic. These functions generate messages of types `SUB`, `UNS`, or `PUB`, respectively, which are sent to all nodes, in case of subscription, or all subscribers of topic t, otherwise.

Propagation of a Message: When node i invokes one of the application functions (Algorithm 1) for topic t, the procedure CO_BROADCAST (line 5 of Algorithm 2) is called, generating a new message of the corresponding type (`PUB`, `SUB`, or `UNS`) which is inserted in the queue of t. Then, a task related to t (Task $START\_MSG\_PROPAGATION$) continuously removes the first message from this queue and starts the broadcast. The next message is removed from the queue only after the reception of acknowledge (message `ACK`) from all current subscribers (per-source FIFO reception order) to whom node i sent the previous message (line 31). The task associated with t is created when node i becomes a new subscriber of the group of topic t (line 11).

Algorithm 1 Functions offered as the interface to the application: node i

```
1:  Init
2:  counter ← 0
3:  ∀t ∈ MAX_TOPICS : view[t] ← ∅

4:  function SUBSCRIBE(topic t)
5:      if ⟨i, SUB, _⟩ ∉ view[t] then
6:          view[t] ← {⟨i, SUB, counter⟩}
7:          CO_BROADCAST(SUB, t, _)
8:          return OK
9:      return NOK

10: function UNSUBSCRIBE(topic t)
11:     if ⟨i, SUB, _⟩ ∈ view[t] then
12:         view[t] ← view[t] ∖ {⟨i, SUB, _⟩}          ▷ removes subscription for t
13:         CO_BROADCAST(UNS, t, _)
14:         return OK
15:     return NOK

16: function PUBLISH(topic t, message data)
17:     if ⟨i, SUB, _⟩ ∈ view[t] then          ▷ only subscribers of t can publish at t
18:         CO_BROADCAST(PUB, t, data)
19:         return OK
20:     return NOK
```

Task $START\_MSG\_PROPAGATION$ for topic t starts the propagation of m, the first message removed from the queue (line 15), by dynamically building a hierarchical spanning tree, rooted at i, composed by the nodes which are either the subscribers of t, in case of messages of type `UNS` or `PUB` or by all nodes, in case of messages of type `SUB` (lines 23-28). For this purpose, node i calls function CHILDREN$(i, t, \log_2 N)$ which renders, for `PUB` and `UNS` messages, the set of the first subscriber nodes of t for each of its clusters (line 26) or the first node of each of i's clusters (line 24) in the case of a SUB message ($t =$ ' $*$ '). These nodes become i's children in the spanning tree and m is sent to them. Upon the reception of m from a node j, by calling function CLUSTER(i, j) (line 42 or 44 depending on the type of message), every child of node i's sends m to its own children in the $s - 1$ clusters, in relation to topic t and the cluster s of i to which j belongs, i.e., $c_{i,s}$. These nodes then become j's children, and so on.

For instance, consider the left side of Figure 3. All nodes are subscribers of t_1, and node p_0, subscriber of t_1, publishes a message m_0 related to t_1 (`PUB` messages). p_0 is the root of the respective spanning tree:

Algorithm 2 Causal broadcast algorithm and delivery executed by node i

1: **Init**
2: $\forall t \in MAX\_TOPICS$: $view[t] \leftarrow \emptyset$; $first\_rec[t] \leftarrow \emptyset$; $not\_delvs[t] \leftarrow \emptyset$; $delv[t] \leftarrow \emptyset$; $br\_queue[t] \leftarrow \emptyset$
3: $msg \leftarrow \emptyset$
4: **create task** $HANDLE\_RECEIVED\_MSG$

5: **procedure** CO_BROADCAST(message_type $type$, topic t, message $data$)
6: NEW(m)
7: $m.type \leftarrow type$; $m.s \leftarrow i$; $m.t \leftarrow t$
8: $m.c \leftarrow counter$; $m.data \leftarrow data$
9: $counter \leftarrow counter + 1$
10: **if** $type = SUB$ **then**
11: **create task** $START\_MSG\_PROPAGATION(t)$
12: $br\_queue[t].insert(m)$

13: **Task** $START\_MSG\_PROPAGATION$(topic t)
14: **loop**
15: $m \leftarrow br\_queue[t].first()$ ▷ block if queue is empty
16: **if** $m.type = PUB$ **then**
17: **if** $\langle i, \_ \rangle \notin first\_rec[t]$ **then**
18: $first\_rec[t] \leftarrow first\_rec[t] \cup \{\langle i, m.c \rangle\}$
19: CO_DELIVER(m)
20: $last\_delvs[t] \leftarrow last\_delvs[t] \setminus \{\langle i, \_ \rangle\} \cup \{\langle i, m.c \rangle\}$
21: $m.cb \leftarrow causal\_barrier[t]$
22: $causal\_barrier[t] \leftarrow \{\langle i, m.c \rangle\}$
23: **if** $m.type = SUB$ **then**
24: $chd \leftarrow \text{CHILDREN}(i, *, \log_2 N)$
25: **else**
26: $chd \leftarrow \text{CHILDREN}(i, t, \log_2 N)$
27: **for all** $k \in chd$ **do**
28: SEND(m) **to** p_k
29: **if** $chd \neq \emptyset$ **then**
30: $acks \leftarrow acks \cup \{\langle \bot, \#(chd), \langle i, t, m.c, \emptyset \rangle \rangle\}$
31: **wait until** $(acks \cap \{\langle \bot, \_, \langle m.s, m.t, m.c, \_ \rangle \rangle\} = \emptyset)$
32: **if** $m.type = UNS$ **then**
33: $msg \leftarrow msg \setminus \{m \mid m.t = t\}$ $not\_delvs[t] \leftarrow \emptyset$
34: $first\_rec[t] \leftarrow \emptyset$; $delv[t] \leftarrow \emptyset$
35: **if** $br\_queue[t] = \emptyset$ **then**
36: exit

```
37: Task HANDLE_RECEIVED_MSG
38: loop
39:     upon receive m from p_j                                          ▷ block if no message
40:         if m.type ≠ ACK then
41:             if m.type = SUB then
42:                 chd ← Children(i, *, Cluster(i, j) − 1)
43:             else
44:                 chd ← Children(i, m.t, Cluster(i, j) − 1)
45:             if chd = ∅ then                                                  ▷ leaf node
46:                 New(m′)
47:                 m′.type ← ACK; m′.s ← m.s; m′.t ← m.t
48:                 m′.c ← m.c; m.data ← ∅
49:                 SendACKs(j, m′)
50:             else                                                          ▷ propagate m
51:                 acks ← acks ∪ {⟨j, #(chd), ⟨m.s, m.t, m.c, ∅⟩⟩}
52:                 for all k ∈ chd do
53:                     Send(m) to p_k
54:         else                                                          ▷ m.type = ACK
55:             k, nb, mem ← k′, nb′, mem′ : ⟨k′, nb′, ⟨m.s, m.c, m.t, mem′⟩⟩ ∈ acks
56:             acks ← acks ∖ ⟨k, nb, ⟨m.s, m.c, m.t, mem⟩⟩
57:             m.data ← m.data ∪ mem
58:             if nb > 1 then
59:                 acks ← acks ∪ ⟨k, nb − 1, ⟨m.s, m.c, m.t, m.data⟩⟩
60:             else if k ≠ ⊥ then                          ▷ All pending ACKs were received
61:                 SendACKs(k, m)

62:         if ⟨i, SUB, _⟩ ∈ view[m.t] then                          ▷ i is subscribed to m.t
63:             if m.type = PUB then
64:                 if (∄⟨m.s, _⟩ ∈ first_rec[m.t]) then
65:                     first_rec[m.t] ← first_rec[m.t] ∪ {⟨m.s, m.c⟩}
66:                 not_delvs[m.t] ← not_delvs[m.t] ∪ {⟨m.s, m.c, m.cb⟩}
67:                 msgs ← msgs ∪ {m}
68:                 CheckDelivery(m.t)                  ▷ received messages may be delivered
69:             else if m.type = ACK then
70:                 view[m.t] ← Update(view[m.t], m.data)
71:             else                                                  ▷ SUB or UNS message
72:                 view[m.t] ← Update(view[m.t], {⟨m.s, m.type, m.c⟩})
73:                 if m.type = UNS then
74:                     first_rec[m.t] ← first_rec[m.t] ∖ {⟨m.s, _⟩}

75: function Update(set_1, set_2)
76:     for all ⟨n_1, _, rc_1⟩ ∈ set_1 do
77:         if (∃ ⟨n_1, _, rc_2⟩ ∈ set_2) then
78:             if rc_2 > rc_1 then
79:                 set_1 ← set_1 ∖ {⟨n_1, _, rc_1⟩}
80:             else
81:                 set_2 ← set_2 ∖ {⟨n_1, _, rc_2⟩}
82:     return set_1 ∪ set_2
```

```
83: procedure CheckDelivery(topic t)
84:     while (∃ ⟨s, c, cb⟩ ∈ not_delvs[t] : CheckCB(t, cb) = true) do
85:         Co_Deliver(m), m ∈ msgs: m.s = s, m.t = t, and m.c = c
86:         not_delvs[t] ← not_delvs[t] ∖ {⟨s, c, cb⟩}
87:         msgs ← msgs ∖ {m}
88:         last_delvs[t] ← last_delvs[t] ∖ {⟨s, _⟩} ∪ {⟨s, c⟩}
89:         causal_barrier[t] ← causal_barrier[t] ∖ cb ∪ {⟨s, c⟩}

90: function CheckCB(topic t, causal barrier cb)
91:     for all ⟨s, c⟩ ∈ cb do
92:         if ( (∃ ⟨s′, c′⟩ ∈ last_delvs[t]: s = s′ and c′ ≥ c)
               or (∃ ⟨s′, c′⟩ ∈ first_rec[t]: s = s′ and c′ > c) ) then
93:             cb ← cb ∖ {⟨s, c⟩}
94:     return (cb = ∅)

95: procedure SendACKs(j, m)
96:     if (⟨i, SUB, _⟩ ∈ view[m.t] and ∄⟨m.s, _⟩ ∈ first_rec[m.t]) then
97:         m.data ← m.data ∪ {⟨i, SUB, c⟩ : ⟨i, SUB, c⟩ ∈ view[m.t]}
98:     Send(m) to p_j
```

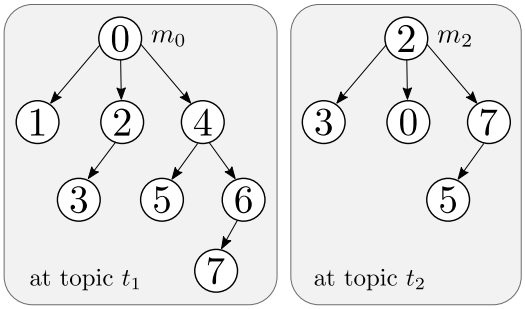


Figure 3: Broadcast trees for two different sources and topics.

m_0 will be sent to the $\log_2 N = 3$ children of p_0 ($\textsc{Children}(0, t_1, 3) = \{1, 2, 4\}$). Upon the reception of message m_0, it is not forwarded by p_1 since $\textsc{Children}(1, t_1, 0) = \emptyset$, while p_2 forwards m_0 to its child p_3, the first subscriber of cluster $c_{2,1}$ ($\textsc{Children}(2, t_1, 1) = \{3\}$). When p_3 receives m_0, as $\textsc{Children}(3, t_1, 0) = \emptyset$, p_3 does not forward m_0 to any node. However, in the case of p_4 ($\textsc{Children}(4, t_1, 2) = \{5, 6\}$), it forwards m_0 to its children $p_5 \in c_{4,1}$ and $p_6 \in c_{4,2}$. Finally, p_6 sends m_0 to p_7.

Consider now a second example, on the right side of Figure 3, where only p_0, p_2, p_3, p_5, and p_7 are subscribers of t_2 and p_2 publishes m_2 related to t_2. In this case, p_2 sends m_2 to each of its child of its $\log_2 N = 3$ clusters that are

also subscribers of t_2: $\textsc{Children}(2, t_2, 3) = \{3, 0, 7\}$ (p_6 is the first node in $c_{2,3}$ but it has not subscribed to t_2). Upon receiving m_0, p_3 does not forward it, because it is already a leaf node in the tree. Node p_0 does not forward it to p_1 since the latter is not a subscriber of t_2. On the other hand, p_7 verifies that in cluster $c_{7,2} = (5, 4)$, p_5 is a subscriber of t_2 ($\textsc{Children}(7, t_2, 2) = \{5\}$), and therefore sends m_2 to p_5 which on its turn does not send it to p_4, because even if p_4 is the first and only node in $c_{5,1}$, it is not a subscriber of t_2. For more details about how to build spanning trees over VCube, see [30].

After forwarding a message m to a child k, node i waits for an `ACK` message from k, which confirms the reception and propagation of m by k. A node will send an `ACK` to its parent node only after it receives itself ACK messages from all its current children related to the topic in question (lines 58-61). `ACK` messages will, thus, be propagated to the root, the source node of m. Eventually the latter receives all the ACK messages it waits for and, in this case, the task related to t removes the next message to be published from the queue associated to the topic t, if there is one. These sequences of `SUB`, `UNS`, or `PUB` and then `ACK` messages from/to the source ensure the per-source FIFO reception order of published messages of the topic.

Reception and Delivery of Messages: When receiving a `PUB` message m of topic t from s (lines 63-68), if node i is a subscriber of t and has not already delivered m, it keeps m in set $msgs$ and both its identification and causal barrier in set $not\_delvs[t]$. If m is the first message received from s to t, i registers it in $first\_rec[t]$ in order to enforce the causal dependencies even under the dynamics of subscriptions. Then, node i verifies, based on direct causal dependencies, which of the previously received messages can be delivered to the application. To this end, node i invokes function $\textsc{CheckDelivery}(t)$ (lines 83-89) which, in it its turn, calls $\textsc{CheckCB}(t, cb)$ in order to check direct dependencies (line 90-94). A message m can be delivered to i only when every message m' on which m causally depends either has already been delivered to i or will never be received by i because *VCube-PS* has not considered i as a subscriber of t during the construction of the spanning tree that broadcast m'. In other words, the first `PUB` message received from s on topic t by i has a higher sequence number than the sequence number of m'. Such a detection of the first message is possible thanks to the $first\_rec_i[t]$ set and the fact that, for the same source, publications of messages of the same topic respect per-source FIFO order.

After delivering m, node i removes it from its pending messages (lines 85-88) and updates its local causal barrier variable (line 89). Note that, since

the delivery of one message m can enable the delivery of other messages that causally depend on m, all remaining non delivered messages are rechecked.
Membership Management: In *VCube-PS*, distributed spanning trees are also used to notify membership changes. When a node i subscribes (resp., unsubscribes) to (resp., from) a topic t, a broadcast `SUB` (resp., `UNS`) message will be received by all (resp., current) subscribers of t. Upon receiving either a `SUB` our `UNS` message, a subscriber of t updates its view of the membership related to t (line 72) by calling function UPDATE(set_1, set_2) (lines 75-82) which merges two membership sets, keeping only the current subscribers.

When a node i subscribes to a topic t, the `ACK` messages related to the `SUB` messages will also gather information about t's membership. Function SENDACKS (lines 95-98) is responsible for sending `ACK` messages. Before forwarding a received `ACK` message to its parent, each subscriber of t includes in the message its current view of t's membership (line 97) merged with the partial membership information coming from its own children (line 57). When receiving all `ACK` messages from its children, the new subscriber i is aware of t's membership.

If node i unsubscribes from topic t, it no longer delivers messages related to the topic (line 33). On the other hand, node i can continue to forward messages related to t to the other subscribers of t in the spanning tree if one of the following situations occurs: (1) there exist subscribers of t that are not aware of i's unsubscription, i.e., they have not received the corresponding `UNS` message from i yet or (2) there are messages queued in i's *br_queue[t]* waiting to be forwarded. Node i also sends `ACK` messages to its parent node in the respective spanning tree. These `ACK` messages are related to published messages that i received and forwarded before leaving t or to messages that satisfy the above-mentioned situations. However, eventually all `ACK` messages will be sent and, thereafter, node i will no more take part in the broadcast of messages related to t. When a subscriber of t receives an `UNS` message related to node i, it removes i from its view of t's membership (line 72) as well as the information about the first message received from i with regard to t (line 74). The latter will be renewed if i rejoins t later.

4. Experimental Results

In order to assess the performance of *VCube-PS* with different configuration scenarios, we conducted experiments with the event-driven PeerSim [25] simulator. In most of the experiments we compare *VCube-PS* to *SRPT*. For

each topic, *SRPT* selects a node to act as the root of the broadcast tree for the respective topic.

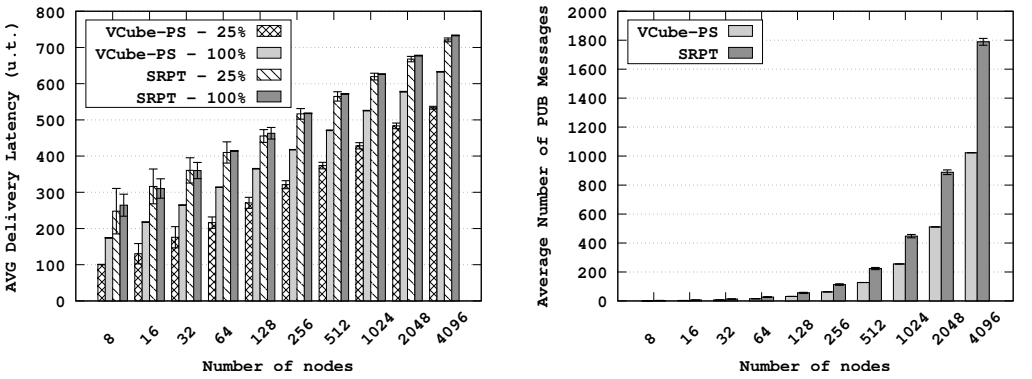
We consider that each message exchanged between two nodes consumes $t_{pc} + t_q + t_t + t_{pp} + t_d$ units of time (*u.t.*). Apart from t_d which represents the time necessary for a subscriber to satisfy all causal dependencies, all other components are based on a packet-switched network delay model [31]: t_{pc} accounts for the processing time of a message by a node; t_q is the time a message must wait in the queue before being transmitted; t_t is the time necessary to transmit all bits of the message into the link; and t_{pp} expresses how long it takes for a message to traverse the link and reach the next hop. Assuming that there is no broadcast feature available in the system, if a message is sent to multiple destinations, a copy of the message is queued for each of the destinations. Based on [32], we set $t_{pc} = t_t = 1$ *u.t.* and $t_{pp} = 100$ *u.t.* (1/100 ratio).

For most experiments, the number of nodes N varies from 8 up to 4096, always a power of two, and each experiment was executed 40 times.

We consider the following metrics for comparison: (1) *Latency*: the time that a published message takes to be received and delivered by all subscribers; (2) *Number of messages*: overall number of `PUB` messages; (3) *Number of messages to be processed by a node*: size of each node's queue; (4) *Size of `PUB` messages*: characterizes the number of direct causal dependencies that `PUB` messages hold; and (5) *Number of false positives*: number of messages received by nodes that act as forwarders of messages of type `PUB`.

4.1. A Single Publisher

This experiment evaluates the impact of the logarithmic properties of *VCube-PS*. A single publisher publishes a single message. Hence, when a subscriber receives the message, there is no delay for delivery. Figure 4(a) shows the delivery latency when the number of nodes of the system varies and either 25% or 100% of them are subscribers. The set of subscribers is randomly chosen following a uniform distribution. In the case of 4096 nodes with 25% of subscribers uniformly distributed, the latency of *VCube-PS* is on average 533 units of time, 26% less compared to the latency of *SRPT* in the same scenario (720 *u.t.*) We remark that when 100% of the nodes are subscribers, *SRPT* has no forwarder and, therefore, the latency of both Pub/Sub systems is always proportional to $\log_2 N$. The only difference in this case is that *SRPT* has an additional hop as the message to be published must be first sent to the root of the only tree that is employed.



(a) Average latency. (b) PUB messages with 25% of subscribers.

Figure 4: Latency and number of messages with a single publisher.

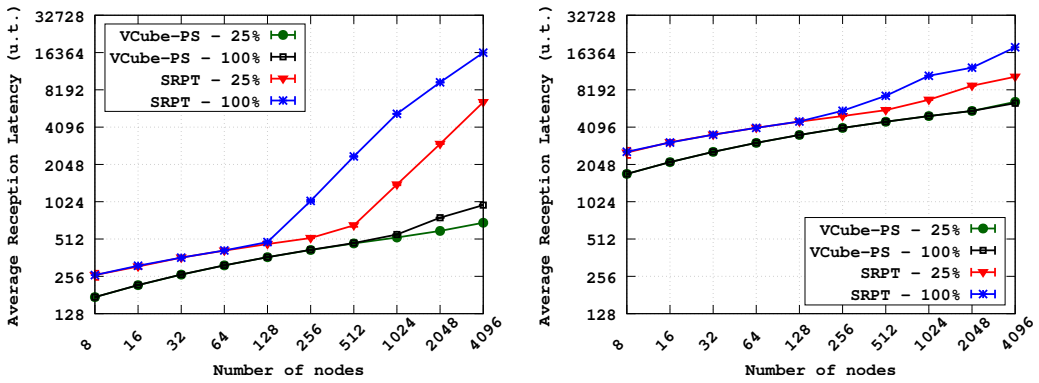
The average number of PUB messages follows the same behavior as shown in Figure 4(b). In the figure, for the two approaches with 25% of the nodes as subscribers, *VCube-PS* always presents the same number for PUB messages, since there is no forwarder in the tree. On the other hand, forwarders in *SRPT* are responsible for up to 2.7 times more messages (for 8 nodes) compared to *VCube-PS*. As the number of nodes increases, this difference is reduced, although *VCube-PS* generates, on average, at least 43% fewer messages than *SRPT* (4096 nodes).

A more detailed analysis of the impact of the number of subscribers in *VCube-PS* and *SRPT* performance is given in the Technical Report [29].

4.2. Multiple Publishers

In these experiments, all nodes are subscribers of a single topic and the number of publishers varies. Each publisher i sends one message at time t_i which is uniformly distributed between $[0, 1000]$ units of time. By having multiple publishers of the same topic, differences in latency will arise from the distribution of the load among the nodes when using one root per publisher (*VCube-PS*) or one root per topic (*SRPT*).

Figure 5 shows in logarithmic scale the average reception latency when the number of nodes of the system varies and either 25% or 100% of them are publishers. Since the ratio between the processing time (t_{pc}) and the propagation time (t_{pp}) has an impact on the load contention, we consider the ratio 1/100 (Figure 5(a)), which is used in all other evaluations of this work,



(a) 1/100 ratio ($t_{pp} = 100$). (b) 1/1000 ratio ($t_{pp} = 1000$).

Figure 5: Reception latency with 25% and 100% of publishers (log. scale).

but also a propagation time which is ten times greater, ($t_{pp} = 1000$ *u.t.*), leading to a ratio 1/1000 (Figure 5(b)).

We can observe in Figure 5(a) that *VCube-PS* presents a maximum increase of 38.8% of the load distribution (4096 nodes and 100% of publishers) when compared to *VCube-PS* with 4096 nodes and 25% of publishers. This happens because even though there are 4 times more messages, they traverse different paths in the network. On the other hand, in *SRPT*, if several messages arrive at the root of the tree at the same time they will be queued before transmission, increasing, thus, the reception latency. For up to 128 nodes, *SRPT* latencies are on average one hop in time higher compared to *VCube-PS*, because in these cases the arrival and output rates of messages are very similar, which avoids contentions. Beyond this number of nodes, the root receives more messages than it can process and transmit per interval of time and starts to saturate. For instance, in comparison with *VCube-PS* with 256 nodes and 100% of publishers, *SRPT* has an average latency 2.48 times greater, and this ratio grows linearly after this point.

Comparing Figures 5(a)(b), the average reception latency increases less in *SRPT* in relation to *VCube-PS* because, with a 1/1000 ratio, it takes longer to receive messages, although the output throughput remains the same.

Table 1 shows the distribution of nodes according to the average size of their sending queues, in a scenario with 1024 nodes, 1/100 ratio, and where all nodes are publishers and subscribers.

The load distribution on the nodes in *SRPT* is uneven when compared

Table 1: Average size of the queue per group of nodes.

# of messages	*# of nodes (VCube-PS)*	*# of nodes (SRPT)*
0	0	512
$(0, 2]$	0	448
$(2, 4]$	0	60
$(4, 8]$	495	3
$(8, 16]$	510	0
$(16, 32]$	19	0
$(32, 4096]$	0	0
$(4096, 8192]$	0	1

to *VCube-PS*: 98% of the nodes in *VCube-PS* have an average load between $(4, 16]$ messages, while 44% of the nodes in *SRPT* have on average between $(0, 2]$ messages in their buffers. In *SRPT*, 50% of the nodes simply do not participate in the routing of any message, because they are leaf nodes of the single tree of the topic and one node (the root) has an average load of 9240 ($\sigma = 4617$) messages, which incurs in high reception latencies.

4.3. Message Order

Besides the published message itself, every PUB message contains its causal barrier, i.e., a list of the direct causal dependencies of the message. Thus, the size of a PUB message increases depending on the number of elements in this list. In order to evaluate the size of the list and the latency due to message ordering, we consider that one node s, chosen randomly, publishes a first message m_s. Upon receiving this message, each node k waits a random interval of time (t_w) before broadcasting message m_k. This situation corresponds to all members of a message discussion group answering a question posted by one of the members. For N nodes, the number of messages is $N^2 - N$ messages. Additionally, we extend this scenario to evaluate the case in which a node k has to wait for at least p messages before broadcasting its own message. To this end, there are $p \geq 1$ nodes that independently broadcast a message each, all in the beginning of the experiment. Just after receiving all these initial messages, any node can publish a message.

Figure 6 groups messages according to the size interval of their causal barriers for *VCube-PS*. When it is necessary to wait for just one message before a node broadcasts its own message, 51.6% of the messages generated in the system have less than 5 preceding messages. More precisely, 19.9% of them have just one causal dependency. On the other hand, if a node waits

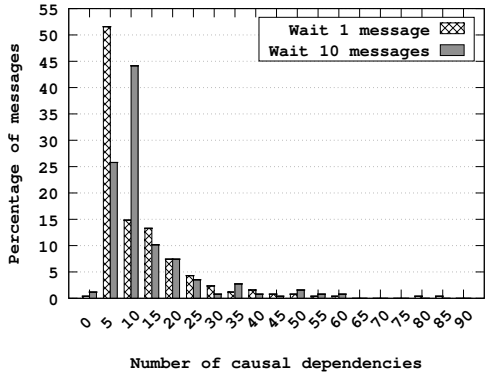


Figure 6: Distribution of causal dependencies for *VCube-PS* with 256 nodes.

for more messages (10 in the case of the figure) before broadcasting its own, a larger number of nodes will have 10 or more direct dependencies. In this case, 35.2% of the messages have size 10 (10 direct dependencies) and 79.7% of them have sizes smaller than 15. However, in both cases, the number of direct dependencies keeps reasonable.

We also evaluated the additional delay imposed by causal barriers before delivering a message. When a node waits for 1 message before broadcasting its own, about 95.1% of the messages are delivered in less than 10 *u.t.* after the message is received (87.2% are delivered with no delay). Only 81 messages (out of 65280) have a delay higher than 50 *u.t.*, with an upper limit of 150 units of time. Increasing the number of the waiting messages to 10, 457 messages wait more than 50 *u.t.* to be delivered (maximum 187), although the number of messages with no delay remains high (84.2%).

4.4. Multiple Topics

As discussed in [10], in real world applications like Twitter, a few topics are related to most of the messages. The authors show that in Twitter, roughly 60% of the topics have only one message published, 83% of them have no more than 5, only 0.15% of the topics are related to more than 1000 messages each. This behavior follows a Zipf-like distribution with a coefficient of 0.825 according to the data provided in the reference. We evaluated *VCube-PS* and *SRPT* with multiple topics. Messages are assigned following both the Zipf-like and uniform distributions. Figure 7 shows results for 256 nodes, 128 topics, and a varying number of messages. Each node publishes a new message on average every 500 *u.t.* for a topic, randomly chosen. Therefore,

messages are uniformly distributed among the publishers, but not necessarily among the topics.

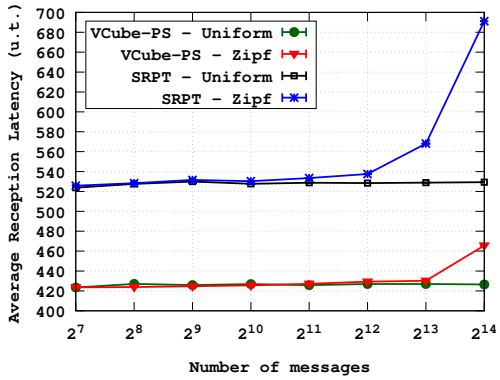


Figure 7: Average reception latency with 256 nodes and 128 topics.

No matter the distribution of messages among the topics, *VCube-PS* always relies on the same root for a given publisher, while *SRPT* does not. This is the reason why the behavior of *SRPT* is the same as *VCube-PS*'s for a uniform distribution of messages. However, when the number of messages sent per node increases beyond a threshold, *VCube-PS* increases the latency due to contention at the source of the messages, i.e., the root of the tree. On the other hand, for the Zipf distribution, *SRPT* has an average reception latency 30.6% higher compared to the uniform distribution (for 2^{14} messages). *VCube-PS* increases latency, on average, only 9.2%.

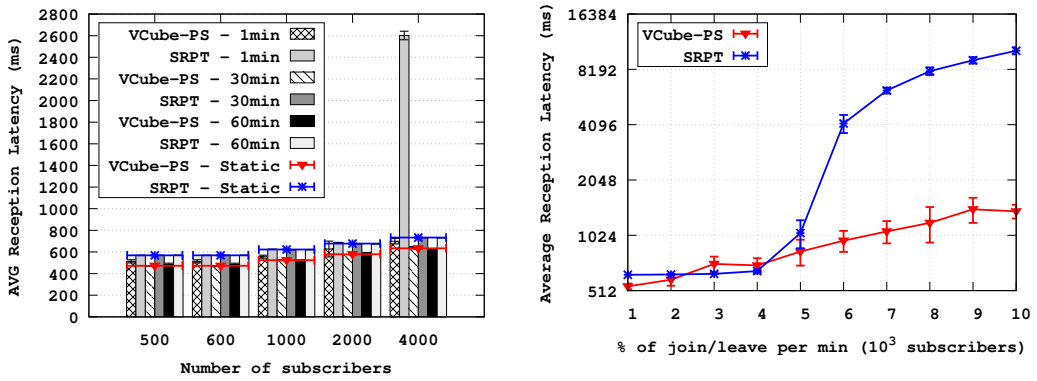
These results confirm that *VCube-PS* is scalable in terms of publishers, while *SRPT* is scalable in terms of topics. However, as noted above, in real scenarios most of the messages are concentrated on a small number of topics.

4.5. Churn Evaluation

In this set of experiments, we evaluate how *SRPT* and *VCube-PS* tolerate membership changes. The parameters used for the evaluation are those proposed by [33], which considers that the time a node keeps connected to a P2P system is heterogeneous (session time) and that the average time ranges from a few minutes up to hours, following a Poisson process. For every node that leaves a given topic, another randomly selected node joins that topic,

thus, always keeping the number of subscribers equals to N_s nodes[1].

For the experiments, we consider one topic and each unit of time represents $1ms$. Every $500ms$, a new message is published by a randomly selected node (uniform distribution). Each simulation corresponds to 120 minutes. Figure 8 presents the average reception latency and standard deviation. It is worth reminding that in *VCube-PS*, every membership change (subscription or unsubscription) generates a new message which is broadcast to all nodes of the system, similarly to a publishing message, while *SRPT* needs to rebuild its per topic single trees. Furthermore, *SRPT* trees often have forwarders (non-subscriber nodes) while in *VCube-PS*, when a node i unsubscribes, it can still receive and forward publications related to the topic for a while (temporary forwarder, see Section 3.3).



(a) Different numbers of subscribers and churn rates.

(b) 1000 subscribers and a varying churn rate.

Figure 8: Average reception latency under churn.

Figure 8(a) summarizes the results with 500, 1000, 2000, and 4000 subscribers. The dynamics of subscriptions were simulated for three different average session times (t_{med}): 1, 30, and 60 minutes. For baseline comparison sake, along with the scenarios with churn, the figure also shows results with static membership. Standard deviation values, although small, are also depicted.

Comparing *VCube-PS* with churn to the static baseline, the former presents

[1] N_s is smaller than the total number of nodes of the simulation in order to have some extra nodes in the churn process while keeping the same hypercube dimension.

average latencies up to 10% higher. In other words, to some extent, *VCube-PS* is sensitive to churn since static membership does not induce false-positives while, with churn, *VCube-PS* has temporary forwarders, responsible for the 10% latency increase. On the other hand, except for 4000 subscribers, *SRPT* latencies vary only up to 1.4% compared to the corresponding static baseline. This stable behavior can be explained as, even in scenarios with no churn, *SRPT* trees have usually non-subscribers (forwarders) and, therefore, the size of their branches does not vary with churn. However, these forwarders are also responsible for the longer *SRPT* tree branches when compared to *VCube-PS* ones, justifying why, for a given churn rate, *SRPT* presents higher latency than *VCube-PS*, independently of the number of subscribers. The highest impact of the churn is observed in *SRPT* with 4000 subscribers and $t_{med} = 1min$, with approximately 46 unsubscriptions and 46 new subscriptions per minute. In this case (high churn rate), the average latency is much higher than the static one (3.56 times), not only because of the presence of false-positives (2.74% of all received `PUB` messages), but also due to contention caused by `SUB` and `UNS` messages. A last interesting observation is that, except for *SRPT* with 4000 subscribers and $t_{med} = 1min$, average latency values of both approaches keep the same behavior and close values for both static and dynamic scenarios.

For the results presented in Figure 8(b) with $N_s = 1000$, the churn rate increases beyond usual values, i.e., it varies from 1% up 10% of the subscribers per minute. In this case, t_{med} varies from $69s$ to $7s$. Note that for the experiments shown in Figure 8(a) with $N_s = 1000$ and $t_{med} = 1min$, the churn rate is approximately 1.1% of the subscribers per minute. Although the higher the churn rate, the greater the number of messages over the network, we can observe in Figure 8(b) that, even if latency increases, *VCube-PS* tolerates quite well the increase in the number of messages: when the churn rate increases 10 times, latency grows in average 2.55 times, false positives represent in average 2.2% ($\sigma = 0.15\%$) of the `PUB` messages, and, in average, messages wait in queue no more than $28.36ms$ ($\sigma = 0.66ms$) before being forwarded. On the other hand, when the churn rate increases, *SRPT*'s single tree is not able to treat and send all the messages in time in order to avoid contention. With churn rate of 4% per minute ($t_{med} = 17s$) and 1000 subscribers, the overall number of sent messages is slightly smaller than that of the scenario with 4000 subscribers and $t_{med} = 1min$ (Figure 8(a)). In both cases, this is the point where *SRPT*'s reception latency starts to suffer from contention. Beyond it, *SRPT*'s single root is unable to treat and forward

messages without queuing them for long periods. For 5% churn rate per minute, messages are kept in queue, in average, $468ms$ ($\sigma = 185ms$) while for 10% churn rate up to $10s$ ($\sigma = 451ms$).

4.6. Broker-based SRPT

For the results presented in this section, we consider a *SRPT* Pub/Sub system based on brokers (e.g., DYNATOPS [5], see Section 5). We call it *SRPT*-B and the previous *SRPT* system was renamed as *SRPT*-S. In *SRPT*-B, the single broadcast tree per topic is composed of nodes that are either brokers (instead of subscribers) or forwarders. Subscribers are directly connected to brokers, according to their locality and/or interests. Each published message for a topic is transmitted over the corresponding tree and each broker, directly sends each received message to the subscribers to which it is connected.

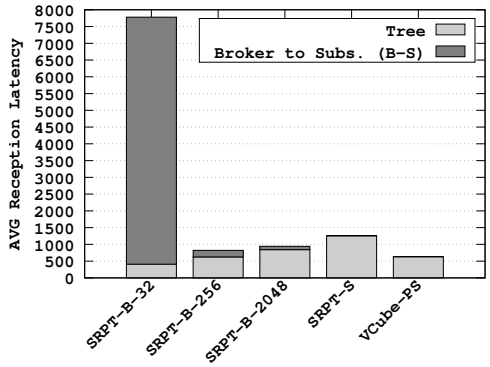


Figure 9: Average reception latency for different approaches and 4096 nodes.

Figure 9 shows the average reception latency for *SRPT*-S, *SRPT*-B, and *VCube-PS*. Publishers are randomly chosen among the subscribers of the topic and send a new message on average every 500 *u.t.*, up to 128 messages.

We defined 3 configurations for *SRPT*-B with different number of brokers: 32, 256, and 2048. The other nodes are subscribers evenly distributed among the brokers: 127, 15, and 1 subscribers per broker. Note that for both *SRPT*-B and *SRPT*-S, forwarders were not employed, i.e., trees are composed only by the respective numbers of brokers or by 4096 subscribers, respectively.

The reception latency of *SRPT*-B is composed of the time to send a message to the brokers (`Tree` in the figure) plus the time for the broker to send the message to the connected subscribers (`B-S` in the figure). On one

hand, we observe that the fewer the number of brokers, the lower the `Tree` reception latency. On the other hand, the fewer the number of brokers, the higher the number of messages per broker forwarded to subscribers, inducing broker-level contention (much like the roots of *SRPT*) and, therefore, the higher the `B-S` reception latency. With 32 brokers, the average `B-S` latency is 4 times higher than the other *SRPT*-B's `B-S` latencies due to high broker-level contention while, with 256 nodes, the load is better distributed.

We also point out that even if *VCube-PS* builds trees with larger heights compared to *SRPT*-B's, it presents lower average reception latency than all *SRPT*-B configurations (22% better for *SRPT*-B with 256 brokers) since it avoids contention by exploiting multiple paths. A last observation is that *SRPT*-B with 2048 brokers has lower reception latency than *SRPT*-S since the latter presents more contention in the root of the tree, which is composed by 4096 subscribers.

5. Related Work

Basically, there exist two models of Pub/Sub systems: *topic-based* [3, 4, 6, 5] and *content-based* [21, 34]. In the first model, subscribers share a common knowledge on a set of available topics and every published message is labeled with a topic. In a topic-based Pub/Sub system, a subscriber can register its interest in one or more topics, and then it receives all published messages related to these topics (e.g., Scribe [3], Bayeux [4], DYNATOPS [5], Dynamoth [6], BeaConvey [9], etc.). In the content-based model [35], messages are structured based on multiple attributes, and subscribers express their interests by specifying constraints over the values of these attributes (e.g., SIENA [36], JEDI [21], BlueDove [37], etc.). This approach offers more flexibility to subscribers for defining their interests, but at the expense of more complex user interfaces and the need for filtering. On the other hand, topic-based systems provide simpler and more efficient implementations and they are usually deployed in contexts where efficient and fast notifications are required.

Similarly to *VCube-PS*, many Pub/Sub systems use tree-based overlays (e.g., Scribe [3], Bayeux [4], Marshmallow [38], DR-Tree [34], DYNATOPS [5], Magnet [7], DRScribe [8], etc.). The advantage of using trees is mainly due to logarithmic guarantees as, for example, reception time of messages with respect to the number of nodes that compose a tree. However, different from *VCube-PS*, most solutions often implement a single multicast tree (usually

one per topic in topic-based systems), statically constructed from the start or rebuilt/updated as nodes join the system. Consequently, every publication should be broadcast from the root of this tree that can become a bottleneck. Moreover, the paths of many of these multicast trees include nodes that are not subscribers but have to forward messages in which they are not interested, thus, the problem of false positives and the need of message filtering (e.g. DR-Tree [34] and Scribe [3]). We also point out that maintenance cost is usually high, specially in presence of churn. Many topic-based Pub/sub systems (e.g. Scribe [3], DYNATOPS [5], Magnet [7], DRScribe [8], etc.) build multicast trees on top of Distributed Hash Table (DHT) overlays (e.g Pastry, CAN). They adopt the *rendezvous* point approach, where a node, responsible for the hash key of a topic name, becomes the *rendezvous* point, i.e., the root of the multicast tree related to the topic. In order to join this tree, a node seeks a DHT path that leads to the root. Hence, nodes in the tree are either subscribers/brokers of the topic or merely forwarders, which are added to the tree because they are there: in the path towards the root. HOMED is a content-based system proposed in [39] that maps nodes to a logical hypercube. However, contrarily to *VCube-PS*, spanning tress are not necessarily rooted at the publisher.

Few Pub/Sub systems ensure message ordering [40, 21, 41, 42], and when they do it is usually total order. Authors in [40] propose a topic-based Pub/Sub system where messages published on different topics are either delivered in the same order to all subscribers or tagged as out-of-order (*weak total order*); while in [41], the task of ordering messages is distributed across sequencer nodes which totally order messages for the same topic. Considering FIFO links, [42] presents a distributed total order protocol for a content-based Pub/Sub system where a broker can decide if a message can be delivered immediately or some consistent delivery order is required. The approach proposed in [43] measures the variations of end-to-end delay of messages, which cause out-of-order messages. Based on the measurements, nodes delay or not the delivery of a message aiming at reducing FIFO delivery order violations. JEDI [21] is a Pub/Sub system that ensures per topic causal order. The latter is implemented by using a *return value*, a message for the receiver to notify the producer that a message was delivered, unlike *VCube-PS*, which does not require these extra messages since causal dependencies of a message are included in the message itself (*causal barriers*). The articles [22, 23] exploit message causal order in Pub/Sub systems. However, neither of them provide a mechanism for assuring causal delivery order of messages for the same topic,

as *VCube-PS*. The first one proposes to causally order messages from different topics while the second one claims to ensure causal order when network partitions are merged, assuming that messages published in each partition are already causally ordered.

6. Conclusion

In this work we presented *VCube-PS*, a distributed topic-based Pub/Sub system. *VCube-PS* propagates information about membership changes and disseminates published messages to the subscribers of a topic using dynamic spanning trees built on top of a hypercube-like topology that presents multiple logarithmic features. While most other Pub/Sub approaches use static trees and *rendezvous* points, *VCube-PS* creates a new spanning tree rooted on the source of every message that is published, without any extra cost, due to *VCube*'s properties. As the spanning trees contain only subscribers of a specific topic, the trees have a shorter height when compared to a per-topic single root tree and, therefore, present lower latencies and employ less messages. Furthermore, *VCube-PS* enforces the causal delivery of messages using causal barriers adapted to cope with the dynamics of the system. Experimental results from simulations on PeerSim confirm benefits of the logarithmic properties of *VCube-PS*. Compared to an approach with one single root per topic, our solution presents the best results under a high publication rate per topic since it intrinsically provides load balancing. Furthermore, *VCube-PS* does not employ permanent forwarders which induce false positives and employs decentralized message broadcast which is efficient in terms of time.

Future directions of our work include adapting the proposed strategy to tolerate node faults. Furthermore, *VCube's* inference rules and causal history, provided by causal barriers, can be exploited in order to combine, without any extra cost, causal related broadcast messages within a single message, reducing, therefore, the number of sent messages and contention over the network, as shown in our paper [44].

References

[1] R. Baldoni, L. Querzoni, A. Virgillito, Distributed event routing in publish/subscribe communication systems: a survey, Tech. rep. (2005).

[2] C. Esposito, D. Cotroneo, S. Russo, On reliability in publish/subscribe services, Comput. Netw. 57 (5) (2013) 1318–1343.

[3] M. Castro, P. Druschel, A. M. Kermarrec, A. I. T. Rowstron, Scribe: a large-scale and decentralized application-level multicast infrastructure, IEEE Journal on Selected Areas in Communications 20 (8) (2002) 1489–1499.

[4] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, J. D. Kubiatowicz, Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination, in: Proc. 11th Int'l Work. Net. Oper. Systems Support for Digital Audio and Video, NOSSDAV '01, ACM, New York, NY, USA, 2001, pp. 11–20.

[5] Y. Zhao, K. Kim, N. Venkatasubramanian, Dynatops: A dynamic topic-based publish/subscribe architecture, in: Proceedings of the 7th ACM International Conference on Distributed Event-based Systems, DEBS '13, ACM, New York, NY, USA, 2013, pp. 75–86.

[6] J. Gascon-Samson, F. Garcia, B. Kemme, J. Kienzle, Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud, in: 35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015, 2015, pp. 486–496.

[7] S. Girdzijauskas, G. V. Chockler, Y. Vigfusson, Y. Tock, R. Melamed, Magnet: practical subscription clustering for internet-scale publish/subscribe, in: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, Cambridge, United Kingdom, July 12-15, 2010, 2010, pp. 172–183.

[8] G. Li, S. Gao, Drscribe: An improved topic-based publish-subscribe system with dynamic routing, in: Proceedings of the 12th International Conference on Web-age Information Management, WAIM'11, 2011, pp. 226–237.

[9] C. Chen, Y. Tock, S. Girdzijauskas, Beaconvey: Co-design of overlay and routing for topic-based publish/subscribe on small-world networks, in: Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems, DEBS '18, 2018, pp. 64–75.

[10] C. Sanli, R. Lambiotte, Local variation of hashtag spike trains and popularity in twitter, PLOS ONE 10 (7) (2015) 1–18.

[11] J. Gascon-Samson, J. Kienzle, B. Kemme, Dynfilter: Limiting bandwidth of online games using adaptive pub/sub message filtering, in: 2015 International Workshop on Network and Systems Support for Games, NetGames 2015, Zagreb, Croatia, December 3-4, 2015, 2015, pp. 1–6.

[12] L. Arantes, M. G. Potop-Butucaru, P. Sens, M. Valero, Enhanced dr-tree for low latency filtering in publish/subscribe systems, in: 2010 24th IEEE International Conference on Advanced Information Networking and Applications, 2010, pp. 58–65.

[13] C. H. Lwin, H. Mohanty, R. K. Ghosh, Causal ordering in event notification service systems for mobile users, in: ITCC (2), 2004, pp. 735–740.

[14] R. Baldoni, M. Raynal, R. Prakash, M. Singhal, Broadcast with time and causality constraints for multimedia applications, in: EUROMICRO, IEEE Computer Society, 1996, pp. 617–624.

[15] C. Plesca, R. Grigoras, P. Queinnec, G. Padiou, J. Fanchon, A coordination-level middleware for supporting flexible consistency in cscw, in: 14th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing, 2006, pp. 6 pp.–.

[16] J. S. Gilmore, H. A. Engelbrecht, A survey of state persistency in peer-to-peer massively multiplayer online games, IEEE Transactions on Parallel and Distributed Systems 23 (5) (2012) 818–834.

[17] M. Ahamad, P. W. Hutto, R. John, Implementing and programming causal distributed shared memory, in: 11th ICDCS, 1991, pp. 274–281.

[18] A. Acharya, B. R. Badrinath, Recording distributed snapshots based on causal order of message delivery, Inf. Process. Lett. 44 (6) (1992) 317–321.

[19] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, C. Hauser, Managing update conflicts in bayou, a weakly connected replicated storage system, in: SOSP, 1995, pp. 172–183.

[20] M. Perrin, A. Mostefaoui, C. Jard, Causal consistency: Beyond memory, in: Proceedings of the 21st ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, PPoPP '16, 2016.

[21] G. Cugola, E. Di Nitto, A. Fuggetta, The jedi event-based infrastructure and its application to the development of the opss wfms, IEEE Trans. Softw. Eng. 27 (9) (2001) 827–850.

[22] H. Nakayama, D. Duolikun, T. Enokido, M. Takizawa, Reduction of unnecessarily ordered event messages in peer-to-peer model of topic-based publish/subscribe systems, in: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA), 2016, pp. 1160–1167. doi:10.1109/AINA.2016.152.

[23] Y. Yamamoto, N. Hayashibara, Merging topic groups of a publish/subscribe system in causal order, in: 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), 2017, pp. 172–177. doi:10.1109/WAINA.2017.96.

[24] E. P. Duarte, Jr., L. C. E. Bona, V. K. Ruoso, VCube: A provably scalable distributed diagnosis algorithm, in: 5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems, ScalA'14, IEEE Press, Piscataway, USA, 2014, pp. 17–22.

[25] A. Montresor, M. Jelasity, Peersim: A scalable p2p simulator, in: 2009 IEEE Ninth International Conference on Peer-to-Peer Computing, 2009, pp. 99–100.

[26] R. Prakash, M. Raynal, M. Singhal, An efficient causal ordering algorithm for mobile computing environments, in: Proceedings of 16th International Conference on Distributed Computing Systems, 1996, pp. 744–751.

[27] K. P. Birman, T. A. Joseph, Reliable communication in the presence of failures, ACM Trans. Comput. Syst. 5 (1) (1987) 47–76.

[28] A. Schiper, J. Eggli, A. Sandoz, A new algorithm to implement causal ordering, in: Proceedings of the 3rd International Workshop on Distributed Algorithms, Springer-Verlag, London, UK, UK, 1989, pp. 219–232.

[29] J. P. de Araujo, L. Arantes, E. P. D. Jr., L. A. Rodrigues, P. Sens, Vcube-ps: A causal broadcast topic-based publish/subscribe system, CoRR abs/1706.08302.
URL `http://arxiv.org/abs/1706.08302`

[30] L. A. Rodrigues, L. Arantes, E. P. Duarte, An autonomic implementation of reliable broadcast based on dynamic spanning trees, in: Dependable Computing Conference (EDCC), 2014 Tenth European, 2014, pp. 1–12.

[31] J. F. Kurose, K. W. Ross, Computer Networking: A Top-Down Approach (6th Edition), 6th Edition, Pearson, 2012.

[32] R. Ramaswamy, N. Weng, T. Wolf, Characterizing network processing delay, in: Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE, Vol. 3, 2004, pp. 1629–1634 Vol.3.

[33] S. Rhea, D. Geels, T. Roscoe, J. Kubiatowicz, Handling churn in a dht, in: Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '04, USENIX Association, Berkeley, CA, USA, 2004, pp. 10–10.
URL http://dl.acm.org/citation.cfm?id=1247415.1247425

[34] S. Bianchi, P. Felber, M. G. Potop-Butucaru, Stabilizing distributed r-trees for peer-to-peer content routing, IEEE Trans. Parallel Distrib. Syst. 21 (8) (2010) 1175–1187.

[35] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131.

[36] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, Achieving scalability and expressiveness in an internet-scale event notification service, in: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, PODC '00, ACM, New York, NY, USA, 2000, pp. 219–227.

[37] M. Li, F. Ye, M. Kim, H. Chen, H. Lei, A scalable and elastic publish/subscribe service, in: 25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings, 2011, pp. 1254–1265.

[38] S. Gao, G. Li, P. Zhao, Marshmallow: A content-based publish-subscribe system over structured p2p networks, in: 2011 Seventh International Conference on Computational Intelligence and Security, 2011, pp. 290–294.

[39] Y. Choi, K. Park, D. Park, Homed: a peer-to-peer overlay architecture for large-scale content-based publish/subscribe system, in: Proceedings of the third International Workshop on Distributed Event-Based Systems (DEBS), Edinburgh, Scotland, 2004, pp. 20–25.

[40] R. Baldoni, S. Bonomi, M. Platania, L. Querzoni, Dynamic message ordering for topic-based publish/subscribe systems, in: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, 2012, pp. 909–920.

[41] C. Lumezanu, N. Spring, B. Bhattacharjee, Decentralized message ordering for publish/subscribe systems, in: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware, Middleware '06, Springer-Verlag New York, Inc., New York, NY, USA, 2006, pp. 162–179.

[42] K. Zhang, V. Muthusamy, H. Jacobsen, Total order in content-based publish/subscribe systems, in: 2012 IEEE 32nd International Conference on Distributed Computing Systems, Macau, China, June 18-21, 2012, 2012, pp. 335–344.

[43] A. Malekpour, A. Carzaniga, G. T. Carughi, F. Pedone, Probabilistic FIFO ordering in publish/subscribe networks, in: Proceedings of The Tenth IEEE International Symposium on Networking Computing and Applications, NCA 2011, August 25-27, 2011, Cambridge, Massachusetts, USA, 2011, pp. 33–40.

[44] J. P. de Araujo, L. Arantes, E. P. D. Júnior, L. A. Rodrigues, P. Sens, A communication-efficient causal broadcast protocol, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, ACM, 2018, pp. 74:1–74:10. doi:10.1145/3225058.3225121.